Extracting models from IDE's[*]

Daniel Langone and Toon Verwaest
Software Composition Group,
University of Berne, Switzerland
http://scg.iam.unibe.ch

Abstract

Systems must co-evolve with their context. Reverse engineering tools are a great help in this process of required adaption. In order for these tools to be flexible, they work with models, abstract representations of the source code. The extraction of such information from source code can be done using a parser. However, it is fairly tedious to build new parsers. And this is made worse by the fact that it has to be done over and over again for every language we want to analyze. In this paper we propose a novel approach which minimizes the knowledge required of a certain language for the extraction of models implemented in that language by reflecting on the implementation of preparsed ASTs provided by an IDE. In a second phase we use a technique referred to as Model Mapping by Example to map platform dependent models onto domain specific model.

1. Introduction

In the field of reverse engineering we need software system analysis tools with different objectives. While such tools are often defined with a specific programming language in mind, generally they only require a fraction of the detail available in it's platform dependent models. This observation has led to the definition of platform independent but domain specific model, such as FAMIX[7]. The rise of metamodels has increased the possibilities for expanding the horizons of reverse engineering techniques by making them automatically applicable to all programming languages for which a conversion to the domain specific model is defined.

It became quickly apparent in the reverse engineering community that building "all-in-one" monolithic tools would be the wrong way to go. Rather it is important to make metamodels, just like models, flexible. Where previously only models could be filed out and imported in other tools, now also metamodels can be moved around freely. For this flexible setup to be possible, a fix-point has to be hardwired, namely a (minimal) metametamodel. In this setting, to make models move around freely we only need to hardwire logic at the higher metametamodel layer. As consequence we also we achieve model portability at all levels of tower. This observation has led to language independent modeling frameworks such as MOF, EMOF[2] and Fame[5].

Up to this point we notice that reverse engineering tools are made flexible in two ways. Firstly because of a domain specific model, tools are not restricted to one specific language anymore. Secondly, the language in which they are written is not fixed up front since as well as models, metamodels can move around freely from environment to environment. The remaining question is where the actual models come from. Up to now, whenever reverse engineers want to convert source code into language independent models, the language had to be mapped manually onto its respective language independent metamodel. This is not only very time-consuming but also requires an expertise in the field of parsing technologies or familiarity with an implementation of an existing tool which sufficiently parses the given language.

This paper proposes a novel way of extracting models from code in an arbitrary language. We combine the knowledge about an implementation of a tool which sufficiently parses the language and the "host-language" in which that tool is implemented.

The paper is structured as follows. In Section 2 we relate our approach to previous work. Section 3 covers the model extraction and model mapping techniques we use in a more detailed way. Finally in section 4 we summarize our results and conclude.

[*] In proceedings of 2nd Workshop on FAMIX and Moose in Software Reengineering (FAMOOSr 2008), pp. 32–35.

2. Language-Independent Metamodel Extraction

Current approaches to model extraction conceptually use a two-step mapping. The first consists of mapping source code onto a language-specific AST, which is done by a parser. This model is still language-specific but more structured than the plain source code. In the second step, a mapping between this language-specific AST and the domain specific model is defined. The first parsers which followed this scheme were fully handcrafted.

A more suitable approach can be found in Parsing by Example [3]. They present a semi-automatic source code to domain specific model mapper. Its implementation is a mix between the well-known *LALR(1) parser*[1][3], *fuzzy parsing*[2][4] and *island grammar*[3] [6]. Although Parsing by Example is a notable improvement over the previously used tools, many new problems arise. First of all, generating correct parsers for generated grammars is already a problematic topic by itself. Secondly it is difficult to detect higher-level structure in flat texts. This is a task which still requires a lot of human guidance.

In an effort to reduce the amount of knowledge of parsers needed, other work has reused existing parsers by hooking into existing tools and only manually implementing the mapping between the AST used in the modified tool and the given domain specific model.

Our approach combines both extraction of models and model-mapping. This would correspond to generation of a specific parser and Parsing by Example. Our approach differs from the existing ones that the techniques do not operate on the same level. First we apply model extraction. In this phase we generate platform dependent models coming from the AST used internally as main representation by the underlying language of the host. This model extraction is done in the first two conceptual phases. The third phase, called Model Mapping by Example, relies on the idea of Parsing by Example. Here we take the extracted platform dependent models and map interactively onto a domain specific model.

The three conceptual phases are:

1) A generic Inferencer tool infers the relevant structure for generating a custom Extractor.
2) The generated Extractor extracts relevant information.
3) We perform "Model Mapping by Example" to map the relevant structure on a given domain specific model.

Phase one and two work on the object space of a running application and can mostly be automated.

1 **L**ook **A**head **L**eft to right parser producing a **R**ightmost derivation with a look ahead of **1** input symbol

2 Partial extraction of a source code model based on syntactical analysis

3 Translation of source code into parts of interest (island) and irrelevant information (water).

3. Steps towards Model Extraction

As previously mentioned, our approach uses three steps to extract the platform dependent models. The first two can be done automated, the third relies on the idea of Parsing by Example. We will refer to it as Model Mapping by Example.

3.1. Inference Phase

During the inference phase we hook into the object space of a running application in order to extract a subgraph of the *class* graph of the application. To do so, it reflects on the runtime structure of the object graph of the application. What we refer to as "relevant structure" is the subgraph of the class graph of the application that models the applications domain. In our case, the application is Eclipse[1] and the relevant domain are AST nodes of Eclipse ASTs representing a source code of the project.

We extract the implementation of an AST of a language as a FM3 compliant representation. To achieve that, we have to select the relevant subgraph of the class graph from the running application. In the host program this information should be available as a subgraph, in our case a subgraph of the Eclipse ASTs. This converted AST will later be used as a platform dependent meta-model. In this step, the user has to manually select the relevant subgraph of the class graph from the running application. For our purpose this is generally a good enough approach since those classes are mostly bundled in a certain package or subclasses of a certain abstract class and therefore should not be hard to find. The challenge is still the selection, as a wrong choice can lead to a wrong platform dependent meta-model for the next step. Therefore we assume that the user has a minimal knowledge about the host application, namely where the abstract AST definition can be found. Eclipse, for example, manages his abstract AST definitions with the help of plugins.

In our approach we automatically map the class subgraphs of the selected subgraph, in our case Java classes, to fame-classes. To extract the needed information we've opted for the the AST visitor provided by Eclipse. Besides that, also fields belonging to the subgraph have to be metadescribed and are mapped to fame-properties. They have to be either of primitive[4] type or of one of the available classes of the subgraph. Other properties should not be metadescribed. Since the whole model graph is not necessarily available upfront by only looking at fields, we also need to include getter methods. An example of such a case is `java.util.HashMap` where values and keys are only accessible through getters. Here the question of which methods to include or exclude automatically pops up. As this step is automated, chosen methods cannot accept arguments. Besides that, the owning class as well as the return type have to be part of the selected subgraph.

This shows that finding correct mappings is not a trivial task. Our main concern during automation is getting out enough of the relevant information. We must decide in an early stage which methods to include or to exclude. We propose three different possible solutions to retrieve this information:

- Follow all getters:

  Pro: No false negatives.

  Contra: Too many false positives, in the worst case scenario the entire class-graph of the host gets included.

- Follow only methods with return type in subgraph:

  Pro: Fewer false positives.

  Contra: Might miss $a_1, a_2 \in \mathit{subgraph}\ f : a_1 \rightarrow a_2$ with $f : (a_1 \in \mathit{subgraph}) \rightarrow (c \notin \mathit{subgraph}) \rightarrow (a_2 \in \mathit{subgraph})$

- Take smallest subgraph so that all $a \in \mathit{subgraph}$ occur as types

  Pro: Fewer false positives. $\forall a_1, a_2 \in \mathit{subgraph}\ \exists f$ with $f : a_1 \rightarrow a_2$.

  Contra: In worst case we have to visit the whole subgraph for one getter method.

The third solution seems the appropriate one, mainly since this step has to be done only once per language in a determined host application. Even if there are false positives, they can be manually excluded in the Model Mapping by Example phase if desired.

3.2. Extraction Phase

In the extraction phase, our generated custom Extractor hooks again into the object space of the same application. This time to extract a subgraph of its *object* graph. In order to do so, it uses the previously extracted information about the runtime structure of the class graph of the application to inspect the runtime object graph. What we refer to as “relevant information” is the subgraph of the running object graph of the application that corresponds to the class subgraph identified in phase one. The extracted subgraph will be the instances that represent the domain data of the application. In our case, a platform dependent models in a portable format conforming to the previously exported platform dependent meta-model.

We convert the concrete syntax tree of a running software system into a readable format. This will make it easier to map it onto a Famix based model. In order to extract such a browsable platform dependent models from the running software system, we use the AST definition from the host application, the AST used by Eclipse in our case. This AST is defined using the format of the underlying language of the host, which is Java in our case. Since we already extracted the AST as a platform dependent meta-model in the previous step, here we map the platform dependent meta-model back onto the underlying language so that we can browse it in a uniform way. Here we use reflection to find the actual Java structure corresponding to the platform dependent meta-model, by looking at the names of the elements of the platform dependent meta-model which are readable since they conform to the FM3 metametamodel of Fame. Once we have re-established the mapping between the Fame-based platform dependent meta-model and the actual Java classes, we can use Fame to understand, browse and export actual instances of the Java classes via the Fame API.

One major difficulty arises in this step. Normally the host program does not provide us directly with the desired AST of the software system. It hides the AST in another object graph with additional host-specific information. This whole graph does not need to be extracted however. After we defined the mapping we need to find the correct starting point of the subgraph in the object graph of the software system. This can be done using a graph search algorithm, as the underlying language of the host application is reflective. Due to that we can iterate through the object graph. We determined to use the Breath First Search algorithm on our object graph. The starting point of the subgraph must have the following format:

- The type of the object must be described in the platform dependent meta-model.
- The type of the node has to be non-primitive.
- The whole subgraph representing the concrete syntax tree has to be reachable from the object.

4 By primitive types we mean all predefined types in the Fame Framework

As previously explained we obtain a platform dependent models, which still has to be transformed before it can be understood by existing reverse engineering tools.

3.3. Model Mapping by Example Phase, Ongoing Work

For the previous two steps there is an implementation in progress, using Eclipse as platform dependent meta-model and platform dependent models provider. The third and last phase is called Model Mapping by Example. The rationale is that it is necessary to map the information resulting from the previous phase onto a language independent one so that it can be used by existing tools. Our previously extracted model is more browsable and contains less information about the specific implementation details of the software system than the actual source code, since we stripped it down so that it only contains the concrete syntax tree. This makes it easier to map it to Famix without prior knowledge of the language of the software system. Besides that we also avoid having to try to generate parsers and therefore eliminate having to try to automatically fix parser generation errors. This step can not be fully automated, because the platform dependent models has no predefined format. It depends on the the structure of the language defined in the software system.

As we directly work with models defined using the Fame framework, we do not need to write a parser. Since Fame provides us with a uniform API for its objects, we can use a straightforward model-visitor to build a semi-automated approach. During this process we would pop up a message as soon as the visitor has found a previously undefined mapping and ask the user to define it. As an aid the system would show possible domain specific model elements onto which the platform dependent models element can be mapped. In this way the visitor would learn the correct mappings. As the name of the approach suggests, the idea from Model Mapping by Example comes from Parsing by Example, although on a higher level. There is no need to map the source code onto the domain specific model, since we already have access to a platform dependent models which we can map onto the domain specific model. This implies that the approach does not generate parser instances, but mapping definitions.

These mapping definitions can be saved in an external file and be reused for later mappings from the same platform dependent meta-model. This gives us the possibility to fall back to already defined mappings from earlier extractions and extending the definitions with the missing ones. Of course this file depends on the used platform dependent meta-model. As such, this file should always be used together with the platform dependent meta-model definition file. The whole procedure becomes more automatic over time as more mapping definitions are available.

4. Conclusion

In this paper we propose a novel way of language independent model extraction. To do so we hook into existing tools in several ways. Firstly we use a flexible model setup as provided by the fame framework. This allows us to move the models from one environment into another. Because of this setup, we can implement the Model Mapping by Example step in a completely different environment then the platform dependent models extraction steps. Secondly, we use the availability of already parsed code in the host system. This concrete syntax tree representation of the software system is a work-around for the parsing problem. Thirdly, we use a technique based on Parsing by Example by mapping the fuzzy model to Famix.

Our approach separates two conceptual phases. The first two steps can be interpreted as model extraction from software system in a running application. The third step can be seen as the second conceptual phase corresponding to Model Mapping by Example. The first conceptual phase is running in a language-related tool. But in fact, our approach is not bound to the fact that those tools are language-related, other than that they have a clean design with a specific part of the application responsible for representing the model of the software system.

References

[1] E. Community. Eclipse, 2008.

[2] S. Ducasse and T. Gîrba. Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 604–618, Berlin, Germany, 2006. Springer-Verlag.

[3] M. Kobel. Parsing by example. Diploma thesis, University of Bern, Apr. 2005.

[4] R. Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1996.

[5] A. Kuhn and T. Verwaest. Fame - a polyglot library for metamodeling at runtime. *Models @ Runtime*, 2008.

[6] L. Moonen. Generating robust parsers using island grammars. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, Oct. 2001.

[7] SCG. Moose analysis technology: Famix, 2008.